A Dispatched Covariant Type System for Numerical Applications in C++

Josef Weinbub*, Philipp Schwaha†, René Heinzl*,
Franz Stimpfl*, and Siegfried Selberherr*

*Institute for Microelectronics, Technische Universität Wien, Gußhausstraße 27-29, 1040 Vienna, Austria
†Shenteq s.r.o., Záhradnícka 7, 81107 Bratislava, Slovak Republic

Abstract. Software development in scientific computing typically deals with a large number of algorithms. An algorithm can have different implementations due to specializations for certain cases. These specializations may result in different return types. If the return types are not known a-priori, meaning, the types cannot be derived during compile-time, run-time techniques are required. This topic is generally referred to as covariant return types and is typically related to the object-oriented programming domain. Common approaches are based on dynamic polymorphism techniques which result in run-time overhead. We present an approach to handle non-a-priori known return types without the need of dynamic polymorphism techniques, hence, increasing run-time efficiency. The approach is discussed in detail based on an example in the field of computational geometry.

Keywords: C++, Boost, Generic Programming, Meta-Programming, Covariant Return Type, Polymorphic Datatype, Dispatch Techniques
PACS: 07.05.Bx, 07.05.Tp, 89.20.Ff

INTRODUCTION

In the field of scientific computing software development is typically confronted with a vast selection of greatly disparate requirements due to the plethora of fields of applications. Unfortunately, the requirements do not necessarily have to be easily reconcilable with each other. It is for such considerations that not only different algorithms, but also different specializations of the same algorithms are implemented. Different implementations vary in the utilization of programming paradigms, which also influences their suitability to various deployments. To support reusability, extendability, and orthogonality the generic programming paradigm [1] is of special concern throughout this work.

From among the manifold reasons for specialized implementations geometrical concerns [2] and those arising from robustness and speed requirements of mesh generation [3] caused the main motivation for the presented investigations.

In general, the interface of generic algorithms is constant, but by using specializations the underlying implementations differ. Such a feature is generally referred to as polymorphism [1]. In the C++ programming language the polymorphic overloading of functions has to be context-independent [4]. This restricts this mechanism to overloading with respect to arguments, but precludes it to encompass return types.

A polymorphic return type can result from dependencies on the input type, as well as from a dependence of an evaluation at run-time, such as accuracy requirements in numerical computations. Therefore the evaluation of the return type must be addressed in the compile-time or the run-time domain, respectively. Several approaches to deal with polymorphic return types have been developed. For example, applying meta-programming techniques [5] enables to derive the return type of an algorithm during compile-time [2]. However, if the internal specialization is selected due to results obtained at run-time, such a technique cannot be used, as the information which specialization should be used is simply unavailable during compile-time.

The object-oriented approach uses the concept of covariant return types to address this issue. Covariance of the return type means that the return type of an overriding method is changed to a type which is related to the return type of the original method [6, 7]. To support different return types a class hierarchy is used. Consequently, all the potential return types must be known a-priori and are introduced into the class hierarchy using inheritance. The run-time selection of the return type is primarily based on virtual function calls.

The problem of polymorphic return types has also been identified in the field of computational geometry [2, 8]. The Computational Geometry Algorithms Library (CGAL) allows encapsulation of the actual result type within a wrapping object [9]. Data can be accessed by trying to assign it from the generic object to variables of other datatypes. The drawback of this approach is that the wrapper object and the polymorphism it provides is resolved dynamically, thus introducing run-time overhead [10].

CP1281, *ICNAAM, Numerical Analysis and Applied Mathematics, International Conference 2010*
edited by T. E. Simos, G. Psihoyios, and Ch. Tsitouras

In contrast to this approach, the Boost Geometry Library uses meta-functions to derive the result type of an algorithm [2]. However, polymorphic return types which depend on run-time evaluations are not considered.

Based on the described situation, we present a typesafe approach for dealing with polymorphic return types without the use of the run-time overhead prone dynamic polymorphism techniques. The potential return types are registered in a heterogeneous compile-time container instead of being forced into a class hierarchy. The polymorphic result type can be tested for a potential datatype, and the data can be extracted using a type safe extraction mechanism.

The following sections apply this technique to a generic implementation of a geometrical predicate. This exemplary application not only shows the use of a polymorphic return type, but it further illustrates the extendibility of the environment with minimized run-time overhead. It further deals with robust implementations in the field of computational geometry. This is realized by using specialized algorithms which base the computations on floating point datatypes of different precision [11]. However, the specializations are not restricted to the given datatypes. Different implementations, either implemented from scratch or externally provided by libraries, can be added to these specializations as well. Thus it results in an orthogonal, extendible approach for library development in the field of computational geometry. Consequently, the applicability of run-time dependent polymorphic return types is shown in potential, high-performance C++ implementations.

THE APPROACH

This section introduces the approach based on an exemplary application case. Figure 1 depicts the implementation design as well as the actual implementation details.

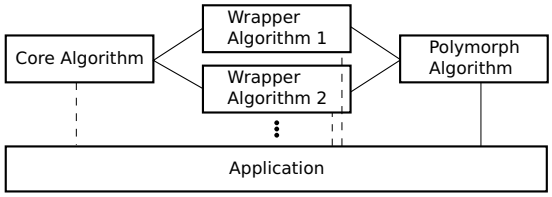

FIGURE 1. The applied generic design of the algorithm implementation. The solid lines denote the different algorithm levels when using the polymorphic implementation. The dashed lines emphasize the generic design, as each level can be accessed separately, should it be desired. The dots denote extendability, as additional specializations can easily be added.

For the remainder of this work, a polymorphic algorithm is understood to have a polymorphic return type, that is, the return type is not uniquely determined from the possible set of return types during compile-time. In the following subsections each part of the depicted implementation design is discussed in detail.

Core Algorithm

A geometrical predicate is used as the core algorithm. More precisely, the two-dimensional, fast but non-robust implementation of an orientation algorithm [3], which is given in the following piece of code.

```
1 template<typename Vector>    typename result_of::value<Vector>::type
2 orient2dfast(Vector const& pa, Vector const& pb, Vector const& pc) {
3   return (pa[0]-pc[0])*(pb[1]-pc[1])-(pa[1]-pc[1])*(pb[0]-pc[0]); }
```

The algorithm is implemented as a free function, where the return type is derived from the value type of the input vector type by the use of a meta-function. The use of a meta-function instead of direct member type access eases the extensibility of the code. This is due to the fact that additional vector types can be supported by providing the value type deriving meta-function.

Wrapper Algorithms

The following code snippet depicts the implementation design of the specialized wrappers. Note, that due to space restrictions, only one wrapper is shown as well as the body of the functor is omitted.

```
template<typename Strategy> struct orient {};
template<> struct orient<tag::accuracy> {
   typedef result_of::value< orient<tag::accuracy> >::type result_type;
   template<typename VectorSequence>
   result_type operator()(VectorSequence const& vector_sequ) {
      // extract, convert, check dimension,
      // call core algorithm, return result                    }};
```

A wrapper is implemented as a specialization based on tags. The tags are empty structs used for identification and dispatching task and are lightweight, even when instantiated. Note, the additional layer of abstraction, as a tag, for example `accuracy`, is related to a specialization based on a high-precision datatype such as `ttmath::Big<8,4>` [11]. This approach depicts the orthogonality of the implementation, as, for example, the actual datatype can be exchanged without influencing the specialized structures. Based on the tags, meta-programming tag-dispatching can be applied which allows to add additional functionality, such as meta-information for example [12]. The result type of the wrapper functors can thus be accessed from outside the functor. Note that additional wrappers can be added to the compile-time tag-dispatch construct. Therefore the extendability of this approach is increased significantly.

Polymorphic Algorithm

The implementation of the polymorphic algorithm is discussed in the following.

```
template<typename ResultTypeSequence>
struct orientation {
   typedef typename result_of::generate_poly_datatype<
      ResultTypeSequence>::type              result_type;
   template<typename VectorSequence, typename Integer>
   result_type operator()(VectorSequence const& vector_sequ, Integer acc_lvl){
      switch(acc_lvl) {
         case 0: return orient<tag::speed>()(vector_sequ);     break;
         case 1: return orient<tag::accuracy>()(vector_sequ); break; ... }}};
```

A polymorphic return type is generated by a meta-function (Line 3-4). This meta-function encapsulates a generation meta-function of the Boost Variant datatype [13], which can be considered the core of this approach regarding polymorphic datatypes. An integer typed accuracy level is used to determine the required algorithm specialization. Additional accuracy levels can be added, by extending the run-time test system, represented by the switch environment.

Application

The following code snippet shows an exemplary usage of the polymorphic orientation algorithm.

```
typedef result_of::value< orient<tag::speed> >::type          speed_result_type;
typedef result_of::value< orient<tag::accuracy> >::type    accuracy_result_type;
typedef boost::fusion::vector<
   speed_result_type, accuracy_result_type >          potential_types;
typedef orientation<potential_types>                  orient_type;
typedef result_of::value<orient_type>::type           polymorph_result;

std::string parse_string("accuracy");
unsigned acc_lvl = parse::eval(parse_string.begin(), parse_string.end());
polymorph_result ply_res = orient_type()(make_vector(pa,pb,pc), acc_lvl);

if(is<accuracy_result_type>(ply_res)) {
   accuracy_result_type val = get<accuracy_result_type>(ply_res); }
```

The actual return types are computed by meta-functions (Lines 1-2). These types are collected in a heterogenous compile-time container [14] and forwarded to the generic, polymorphic functor implementation (Lines 3-5).

Note, that as with the common covariant return types approach, the potential types are required to be known a-priori. The polymorphic return type, based on the Boost Variant datatype, of the algorithm is computed by an external meta-function (Line 6). A parser is used to depict the run-time applicability of the presented implementation (Lines 8-9). Note that a representative test string is used to model the application scenario. Typically this string would be provided by some sort of input stream. The parsing facility investigates the content of the string and sets an accuracy level accordingly. This exemplary application emphasizes the run-time capabilities of the presented approach, as a parser investigates a string solely during run-time. This level is forwarded to the functor, additionally with a collected point sequence, denoted by `data` (Line 10). Since the points are encapsulated in a container, the functor interface is decoupled from the number of input points. This approach increases the genericity of code, as further specializations for different sets of point types can be added. The polymorphic result is tested to match a certain type (Line 12) and the data is extracted in case it matches (Line 13). Type safety is maintained, since the type used for testing is the same as the type used for actual data extraction. The implementations of the functions `is()` and `get()` provide convenient layers by encapsulating access to the underlying Boost Variant datatype. Thus the run-time overhead of determining which of the possible datatypes is actually in use, is, due to the Boost Variant mechanisms, which employ meta-programming techniques instead of dynamic polymorphism techniques, kept to a bare minimum.

CONCLUSION AND OUTLOOK

The issue of polymorphic return types in case of a generic application in the field of computational geometry has been discussed. Compile-time and run-time techniques have been depicted, as well as advantages and disadvantages have been outlined. An approach has been presented, which is capable of dealing with run-time switchable polymorphic return types. An extendible exemplary implementation has been discussed as well as the implementation details of the polymorphic return type approach has been given. It is important to note, that our approach has a broad applicability. The depicted example from the field of robust computational geometry, which is implemented using high-precision datatypes, introduced the need of an automatic high-precision library selection environment. This is of special interest, as there are already several multi-precision libraries available [15, 16]. The goal would be to provide a facility which automatically determines which of these libraries is best suited for a given problem. Naturally, the process of library selection should be based on run-time evaluation. Therefore, the presented polymorphic return type is of great interest in that context.

ACKNOWLEDGMENTS

This work has been supported by the European Research Council through the grant #247056 MOSILSPIN and by the Austrian Science Fund FWF, project P19532-N13.

REFERENCES

1. R. Heinzl, *Concepts for Scientific Computing*, Dissertation, Technische Universität Wien (2007).
2. B. Gehrels, B. Lalande, and M. Loskot, "Generic Programming for Geometry," BoostCon'10, 2010.
3. J. R. Shewchuk, "Robust Adaptive Floating-Point Geometric Predicates," in *SCG '96: Proc. Symp. on Comp. Geometry*, ACM Press, 1996, pp. 141–150, ISBN 0-89791-804-5.
4. B. Stroustrup, *C++ Programming Language, The (3rd Edition)*, Addison-Wesley, 1997.
5. D. Abrahams, and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*, Addison-Wesley, 2004, ISBN 0321227255.
6. C. Allison, "What's New in Standard C++?," C/C++ Users Journal, 1998, vol. 16, pp. 69–81.
7. G. Castagna, "Covariance and Contravariance: Conflict Without a Cause," in *Trans. on Program. Lang. and Syst.*, ACM, 1995, vol. 17, pp. 431–447, ISSN 0164-0925.
8. Computational Geometry Algorithms Library(CGAL) (1998-2010), `http://www.cgal.org`.
9. A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr, "On the Design of CGAL, a Computational Geometry Algorithms Library," Software: Practice and Experience, John Wiley & Sons, 2000, vol. 30, pp. 1167–1202.
10. K. Driesen, and U. Hölzle, "The Direct Cost of Virtual Function Calls in C++," ACM Press, 1996, vol. 31, pp. 306–323, ISSN 0362-1340.
11. TTMath - Bignum C++ Library (2010), `http://www.ttmath.org`.
12. D. Abrahams, and D. Gregor, Generic Programming in C++: Techniques (2001), `http://www.generic-programming.org/languages/cpp/techniques.php`.
13. Boost Variant (2002-2003), `http://www.boost.org/libs/variant`.
14. Boost Fusion 2.0 (2001-2007), `http://www.boost.org/libs/fusion`.
15. *MPFR Library* (2000-2010), `http://www.mpfr.org`.
16. *GMP Library* (2000-2010), `http://gmplib.org`.